Implementing the finite element method

Computer techniques for modellers

David Ham

David.Ham@imperial.ac.uk

Imperial College London

An equation!

$$\nabla^2 \psi = f(\mathbf{x})$$

on some domain Ω with boundary condition $\nabla \psi = 0$.

An equation!

$$\nabla^2 \psi = f(\mathbf{x})$$

on some domain Ω with boundary condition $\nabla \psi = 0$.

Multiply by a test function and integrate:

$$\int_\Omega N \nabla^2 \psi dV = \int_\Omega N f(\mathbf{x}) dV$$

An equation!

$$\nabla^2 \psi = f(\mathbf{x})$$

on some domain Ω with boundary condition $\nabla \psi = 0$.

Multiply by a test function and integrate:

$$\int_\Omega N \nabla^2 \psi dV = \int_\Omega N f(\mathbf{x}) dV$$

Integrate by parts.

$$-\int_\Omega \nabla N \nabla \psi dV + \int_\Gamma N \nabla \psi dA = \int_\Omega N f(\mathbf{x}) dV$$

An equation!

$$\nabla^2 \psi = f(\mathbf{x})$$

on some domain Ω with boundary condition $\nabla \psi = 0$.

Multiply by a test function and integrate:

$$\int_\Omega N \nabla^2 \psi dV = \int_\Omega N f(\mathbf{x}) dV$$

Integrate by parts.

$$-\int_\Omega \nabla N \nabla \psi dV + \int_\Gamma N \nabla \psi dA = \int_\Omega N f(\mathbf{x}) dV$$

An equation!

$$\nabla^2 \psi = f(\mathbf{x})$$

on some domain Ω with boundary condition $\nabla \psi = 0$.

Multiply by a test function and integrate:

$$\int_\Omega N \nabla^2 \psi dV = \int_\Omega N f(\mathbf{x}) dV$$

Integrate by parts.

$$-\int_\Omega \nabla N \nabla \psi dV = \int_\Omega N f(\mathbf{x}) dV$$

A single element

If we have n unknowns and g gauss points per element then for each element the left hand side becomes a local $n \times n$ matrix:

$$\mathrm{M}(i,j) = -\sum_{k \in g} \nabla N_i(\mathbf{x_k}) \nabla M_j(\mathbf{x_k}) |J^{-1}| w(k)$$

A single element

If we have n unknowns and g gauss points per element then for each element the left hand side becomes a local $n \times n$ matrix:

$$\mathrm{M}(i,j) = -\sum_{k \in g} \nabla N_i(\mathbf{x_k}) \nabla M_j(\mathbf{x_k}) |J^{-1}| w(k)$$

Then if ele_Ψ is the n-vector of the global numbers of the unknowns in the current element then we can write:

$$\mathrm{A}(\mathrm{ele}_\Psi, \mathrm{ele}_\Psi) = \mathrm{A}(\mathrm{ele}_\Psi, \mathrm{ele}_\Psi) + M$$

The right hand side

If we have n unknowns and g gauss points per element then for each element the right hand side becomes a local n matrix:

$$\mathbf{r}(i) = \sum_{k \in g} N_i(\mathbf{x_k}) f(\mathbf{x_k}) |J^{-1}| w(k)$$

The right hand side

If we have n unknowns and g gauss points per element then for each element the right hand side becomes a local n matrix:

$$\mathbf{r}(i) = \sum_{k \in g} N_i(\mathbf{x_k}) f(\mathbf{x_k}) |J^{-1}| w(k)$$

Again if ele_Ψ is the n-vector of the global numbers of the unknowns in the current element then we can write:

$$\mathbf{b}(\mathrm{ele}_\Psi) = \mathbf{b}(\mathrm{ele}_\Psi) + \mathbf{r}$$

Finally! A matrix equation

Having assembled a left hand side matrix and a right hand side vector we have:

$$\mathrm{A}\boldsymbol{\Psi} = \mathbf{b}$$

A test case

As a simple 2D test case we take:

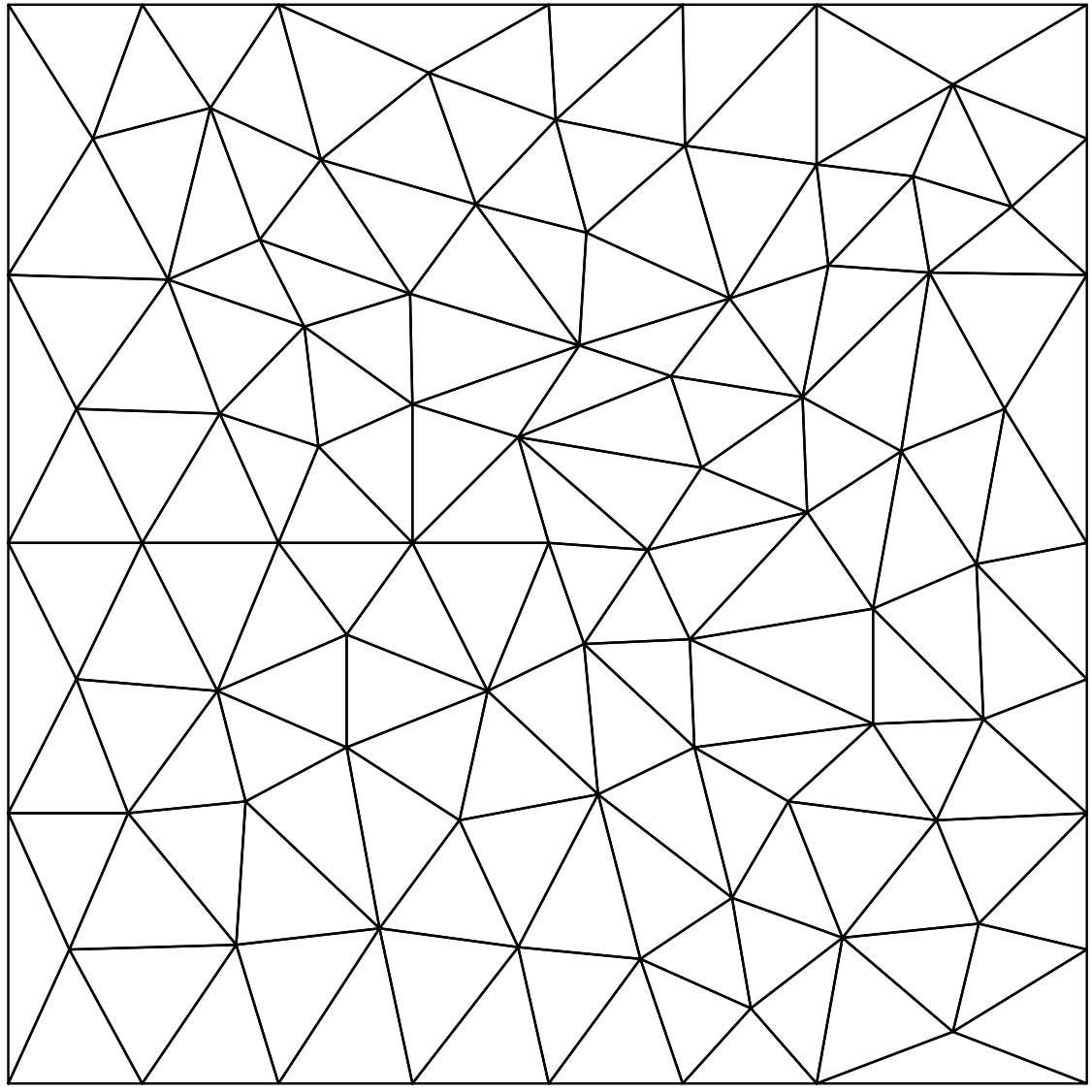
$$f(x, y) = -8.0\pi^2 \cos(2\pi x) \cos(2\pi y)$$

Which has analytic solution:

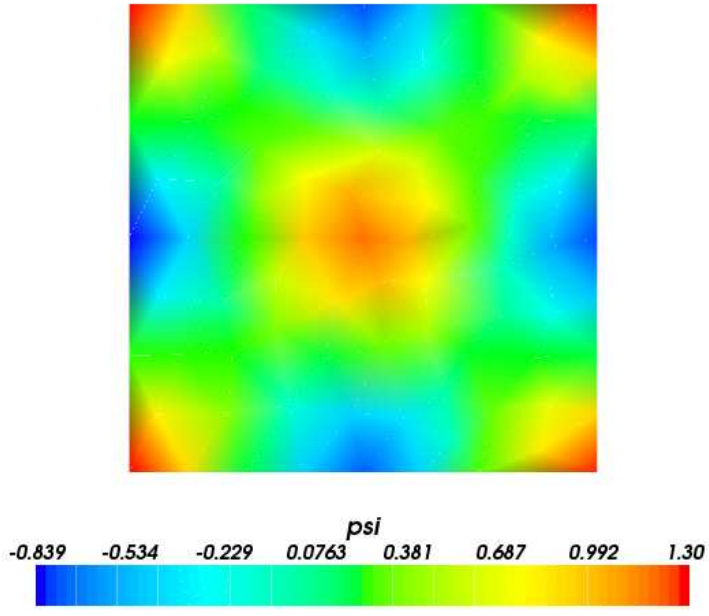
$$\Psi(x, y) = \cos(2\pi x) \cos(2\pi y) + C$$

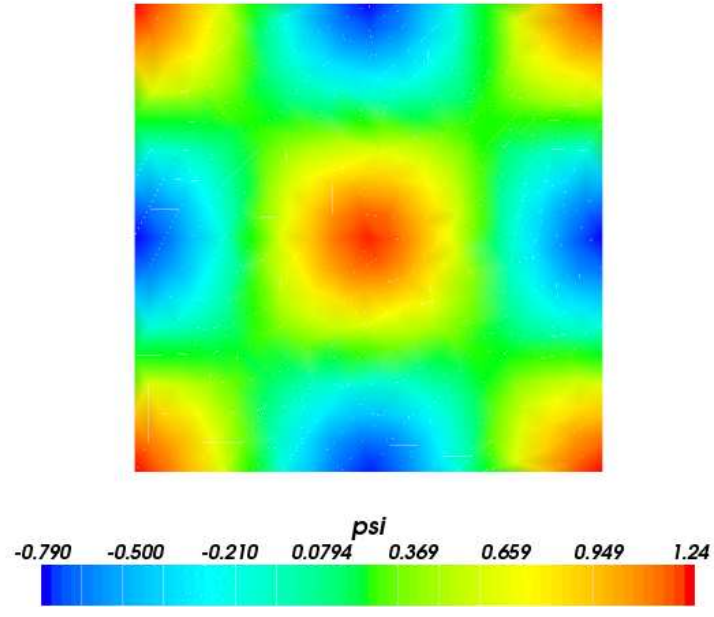
for an arbitrary constant C.

The mesh

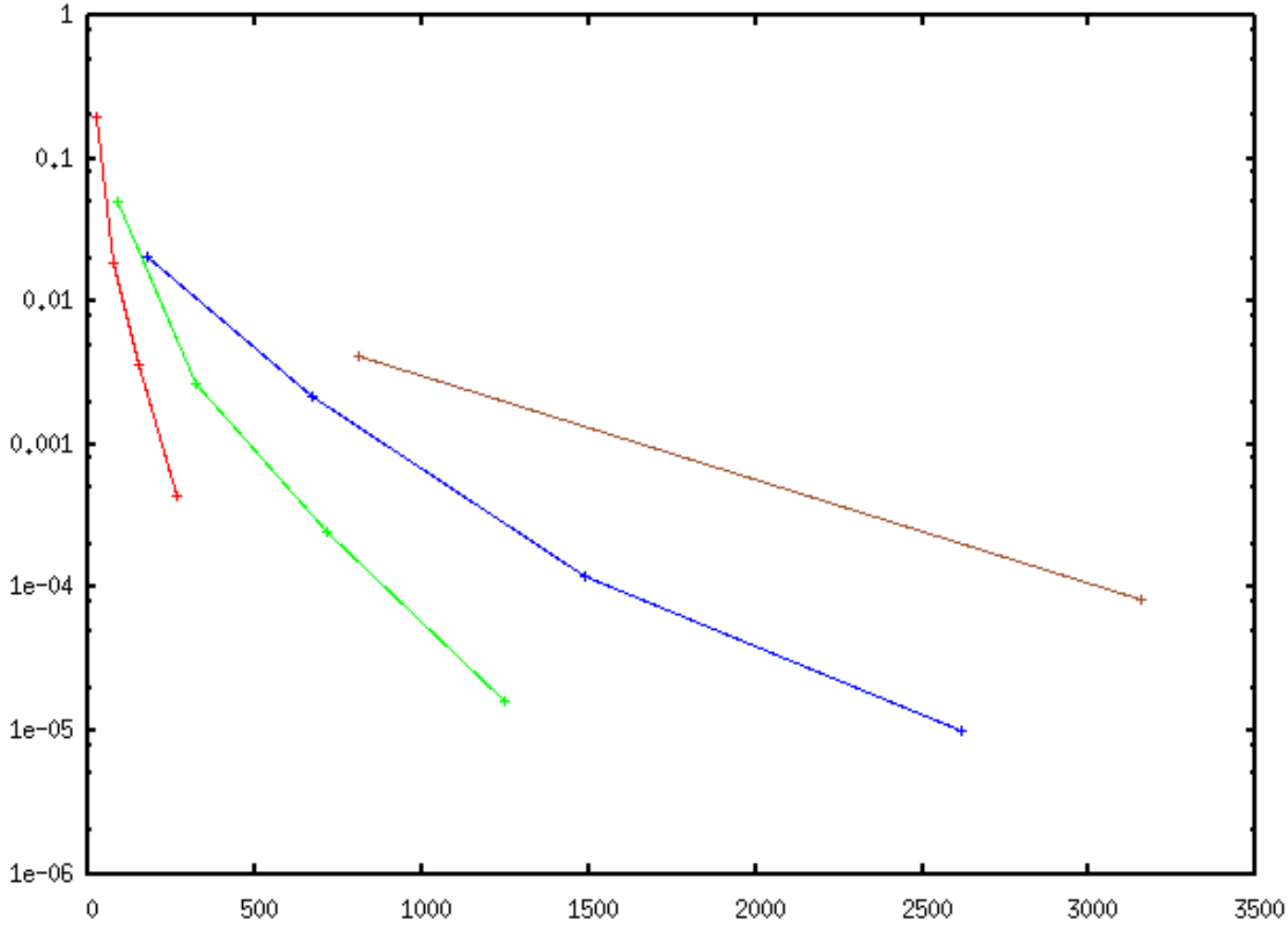


Linear and quadratic solutions





Error for h and p refinement



Error against degrees of freedom. Lines link results on the same mesh.

Pointers

Pointers enable one name to be associated with different pieces of memory at runtime.

```
real, dimension(10), target :: array
real, dimension(:), pointer :: ptr1, ptr2

forall (i=1:10) target=i

ptr1 => array
ptr2 => ptr1

print *,ptr2(3) ! This prints 3
ptr1=0
print *,ptr2(3) ! This prints 0
print *,associated(ptr1) ! This is .true.
print *,associated(ptr1,ptr2) ! This is .true.
print *,associated(ptr2,array) ! This is .true.
ptr2=>null()
print *,associated(ptr2,array) ! This is .false.
```

Pointers and allocate

```
real, dimension(:), allocatable, target :: array
real, dimension(:), pointer :: ptr1, ptr2

allocate(ptr1(10), target(10))

ptr2=>array
deallocate(ptr2) ! This is not allowed!

ptr2=>ptr1(3:5) ! ptr2 is associated with an array section.
deallocate(ptr2) ! This is not allowed!

ptr2=>ptr1
deallocate(ptr2) ! This is allowed.

print *, associated(ptr1) ! This is .false.
print *, allocated(ptr1) ! This is not allowed!
```

Pointer association status

Pointers can be:

- associated
- not associated
- undefined

Undefined is a miserable state for a pointer to be in and should be avoided at all costs!

Derived data types

```
program daily_libel
  implicit none
  type person_type
     character(len=256) :: name="", job=""
     integer :: age
  end type person_type
  type(person_type) :: person
  person=person_type("J. Random Hacker","Geek",34)
  call defame(person)

contains
  subroutine defame(victim)
    type(person_type), intent(in) :: victim
    print '(a,i0,a)', trim(victim%name)//" (",victim%age,"), a "&
          &//trim(victim%job)&
          &//" has been caught interfering with sheep."
    end subroutine defame
end program daily_libel
```

A data type for finite elements

```
type element_type
   ! Type to encode shape and quadrature information for
   ! an element.
   integer :: dim ! 2d or 3d?
   integer :: loc ! Number of nodes.
   integer :: ngi ! Number of gauss points.
   ! Shape functions: n is for the primitive function, dn is for
   ! partial derivatives.
   real, pointer :: n(:,:)=>null(), dn(:,:,:)=>null()
   ! Link back to the node numbering used for this element.
   type(ele_numbering_type), pointer :: numbering=>null()
   ! Link back to the quadrature used for this element.
   type(quadrature_type), pointer :: quadrature=>null()
end type element_type
```

Making elements

```
g=make_quadrature(loc=4, dimension=3, degree=quad_degree)
m=make_element_shape(loc=4, dimension=3, degree=degree, quad=g)
n=make_element_shape(loc=4, dimension=3, degree=n_degree, quad=g)

if (fe_method==GEO_DG) then
    ! Faces are triangles.
    g_f=make_quadrature(loc=3, dimension=2, degree=quad_degree)
    m_f=make_element_shape(loc=3, dimension=2, degree=degree, &
        quad=g_f)
    n_f=make_element_shape(loc=3, dimension=2, degree=n_degree, &
        quad=g_f)
end if
```

Deallocating elements

Always write an allocate and deallocate for variable sized data types.

```
call deallocate(m) ! This is not intrinsic deallocate!
call deallocate(n)
call deallocate(g)
if (fe_method==GEO_DG) then
   call deallocate(m_f)
   call deallocate(n_f)
   call deallocate(g_f)
end if
```

Transform to physical coordinates

```
subroutine transform_to_physical(X, n, m, dn_t, dm_t, detwei)
Reference velocity element: type(element_type), intent(in) :: n
Reference pressure element: type(element_type), intent(in), &
optional :: m
! Column n of X is the position of the nth node. (dim x n%ngi)
real, dimension(:,:), intent(in) :: X
! Pressure shape derivatives in
! physical coordinates. (n%loc x n%ngi x dim)
real, dimension(:,:,:), intent(out), optional :: dm_t
! Velocity shape derivatives in
! physical coordinates. (n%loc x n%ngi x dim)
real, dimension(:,:,:), intent(out), optional :: dn_t
! Quadrature weights for physical coordinates.
real, dimension(:), intent(out), optional :: detwei (n%ngi)
```

Building a mass matrix

```
function shape_shape(shape1, shape2, detwei)
  ! For each node in each element shape1, shape2 calculate the
  ! coefficient of the integral int(shape1shape2)dV.
  ! Detwei is the gauss weights transformed by the coordinate
  ! transform from real to computational space.
  ! In effect, this calculates a mass matrix.
  type(element_type), intent(in) :: shape1, shape2
  real, dimension(shape1%ngi), intent(in) :: detwei
  real, dimension(shape1%loc,shape2%loc) :: shape_shape

  forall(iloc=1:shape1%loc,jloc=1:shape2%loc)
     ! Main mass matrix.
     shape_shape(iloc,jloc)=&
          dot_product(shape1%n(iloc,:)*shape2%n(jloc,:),detwei)
  end forall

end function shape_shape
```

Other derived data types in Fluidity

quadrature_type Element quadrature

ele_numbering_type Local node indices

csr_matrix Sparse matrix

scalar_field , vector_field Whole field descriptors